AUEB M.Sc. in Data Science

- Course: **Deep Learning**
- Semester: Spring 2020
- Instructor: Prof. P Malakasiotis
- Author: S. Politis (p3351814)
- Homework: 1

Introduction and scope

Homework description (as given by instructor)

Submit a report (max. 5 pages, PDF format) for the following machine learning projects that follow. Explain briefly in the report the architectures that you used, how they were trained, tuned, etc. Describe challenges and problems and how they were addressed. Present in the report your experimental results and demos (e.g., screenshots) showing how your code works. Do not include code in the report, but include a link to a shared folder or repository (e.g. in Dropbox, GitHub, Bitbucket) containing your code. The project will contribute 30% to the final grade.

Given an image of a fashion item, build a deep learning model that recognizes the fashion item. You must use at least 2 different architectures, one with MLPs and one with CNNs. Use the Fashion-MNIST dataset to train and evaluate your models. More information about the task and the dataset can be found at https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/ (https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/)

Goals description

The scope of the assignment is to lead the student to a firm understanding of key tasks related to creating, training and evaluating the performance of some neural network architectures, specifically *MLP*s and *CNN*s, for the task of image classification. The assignment is not designed with the the end goal to produce the best classifier, rather to allow the student to demonstrate techniques to tweak the performance of neural networks.

With this goal in mind, the assignment will focus on the following tasks:

- Specifying the task at hand: this is an essential step which, given the nature of the problem, will define its parameters, such as loss function to use.
- Ingestion and preprocessing of input data so as to be in the correct input form for the neural network architectures that will be developed.
- Sound, both theoretically and practically, splitting of train / dev / test sets.
- Sound definition of the metrics required to assess the model performance.
- Implementation and presentation of a few *MLP* and *CNN* architectures, their relative capacity, performance and possible shortcomings. Description of the weights initialization, activation functions, regularization techniques and optimization algorithm selected will also be provided and assesed.
- Hyperparameter tuning for the best architecture selected, after manual implementation and assesment.

Setup and environment

Deep learning is well-known to benefit from *CUDA*-accelerated infrastructure, since matrix operations are orders of magnitude faster on GPU than CPU hardware. The development machine is equiped with an *Nvidia GTX1060 6GB RAM* GPU and the environment characteristics are as follows:

Param	Value
Tensorflow version	2.1.0
Built with CUDA	True
GPU	PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')

A note about random seed: we have set the *Numpy* and *Tensorflow* seed so that stochasticity in repeated experimentation is removed and results are reproducible, especially when performing hyperparameter tuning.

Specifying the task

The task of this assignment is to preform multiclass classification on the input set, based on the training set labels provided (10 in total).

Documenting the code

In order to have useful abstractions, we have created the following *Python* packages:

- Data: implements the functions for ingesting the input data (Fashion-MNIST images and classes)
- Model: implements the classes and functions for constructing *MLP*s and *CNN*s
- Evaluation: helper methods to evaluate the performance of the models
- Visualization: implements all functions for drawing dataset images, classes distributions and *TensorFlow* history object values (*loss*, *val_loss*, *accuracy*, *val_accuracy* etc.) as graphs

Note that we will be using the *TensorFlow functional API* to build our models, the reason being that it is more flexible for constructing neural network architectures. This flexibility is a non-issue for this assignment, however the knowledge obtained may prove valuable in the future.

Data ingestion

Initial dimensions of ingested data are as follows:

Dataset	Value
X_train	(60000, 784)
y_train	(60000,)
X_test	(10000, 784)
y_test	(10000,)

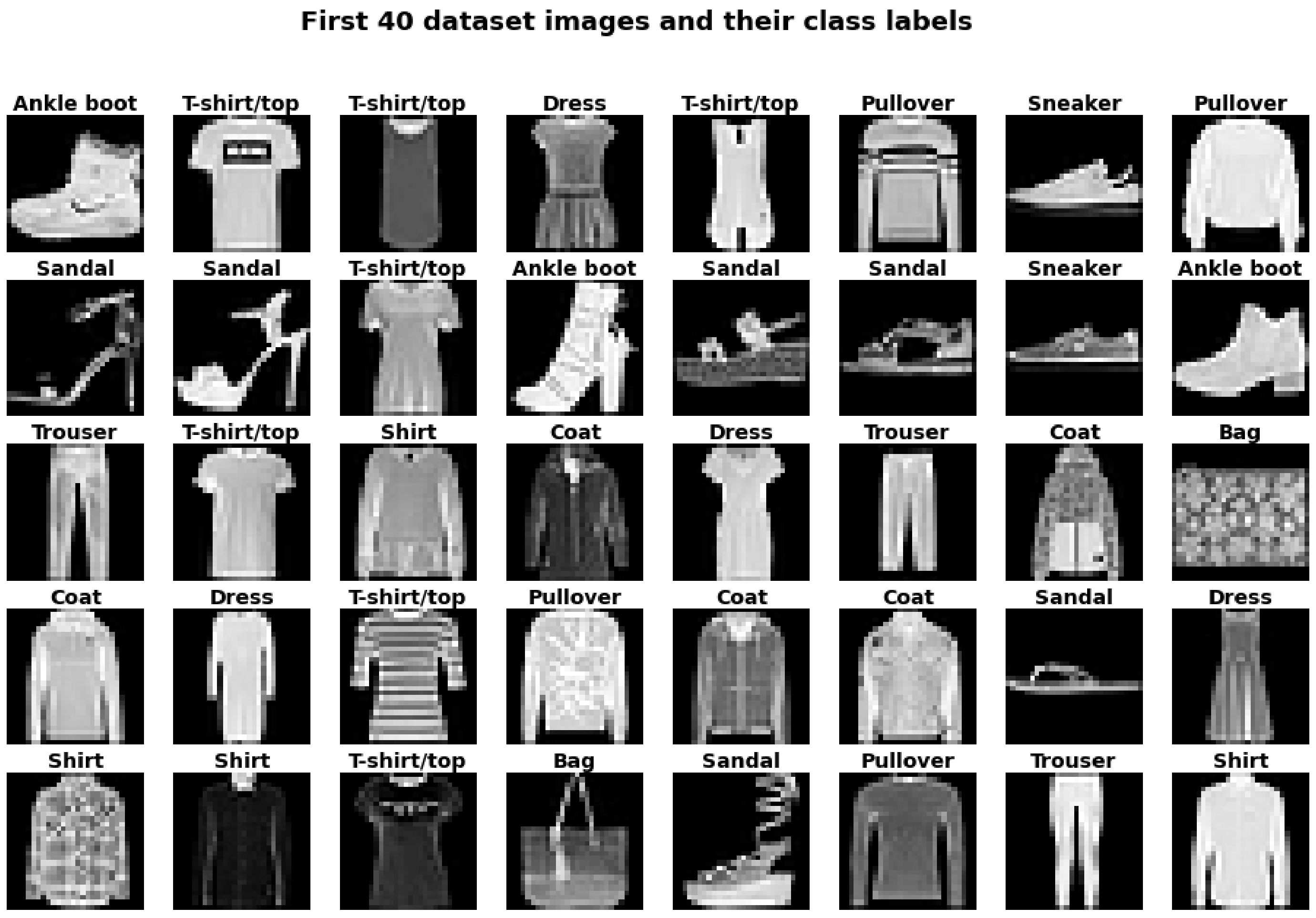
We observe that there exist 60000 training examples and 10000 test examples. The data features are rolled-out in arrays of shape $28 \times 28 = 784$, 28×28 being the dimensions of input images.

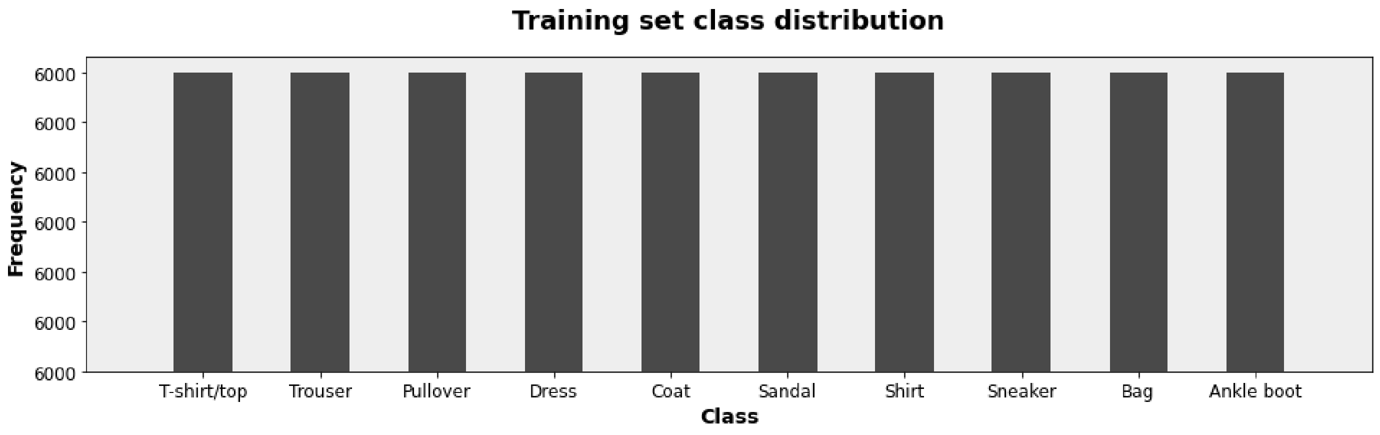
Data augmentation

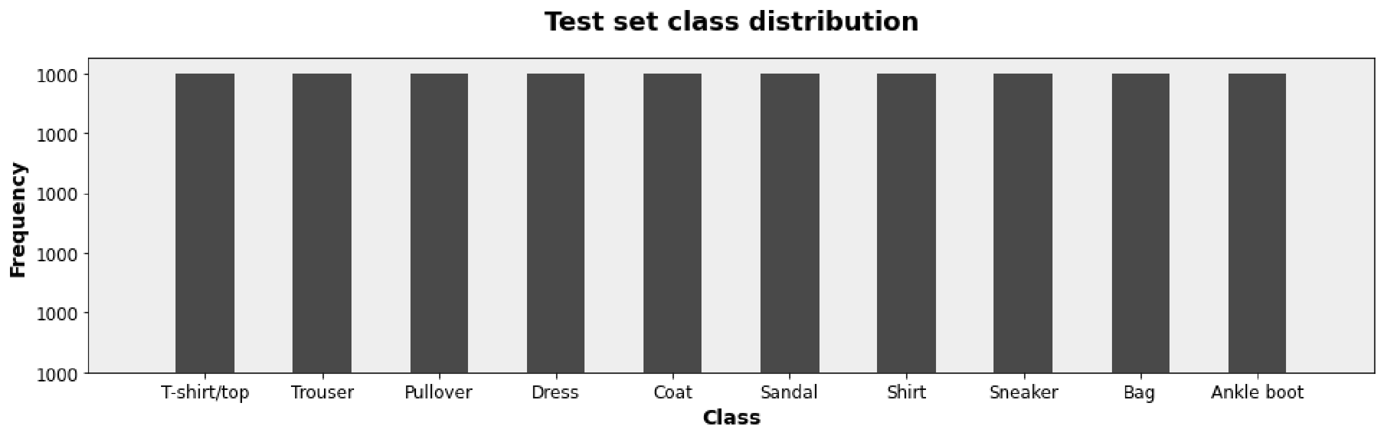
Non-applicable for the scope of thios assignment, as volume of examples seem to be adequate for the task.

EDA

Brief exploratory data analysis follows below and focuses mainly on getting a feel of the data, as well as making sure that no class imbalance problem exists.







We observe that the distribution of classes in both the training and the test set are uniformly distributed, hence the dataset does not present a class imbalance problem.

Data preprocessing

Pixel values normalization

The purpose of normalization is to fit the feature space to $[0, 1]$ because unscaled input variables can result in a slow or unstable learning process. Normalization was achieved simply by dividing each pixel value by 255 (the maximum pixel value for grayscale images).

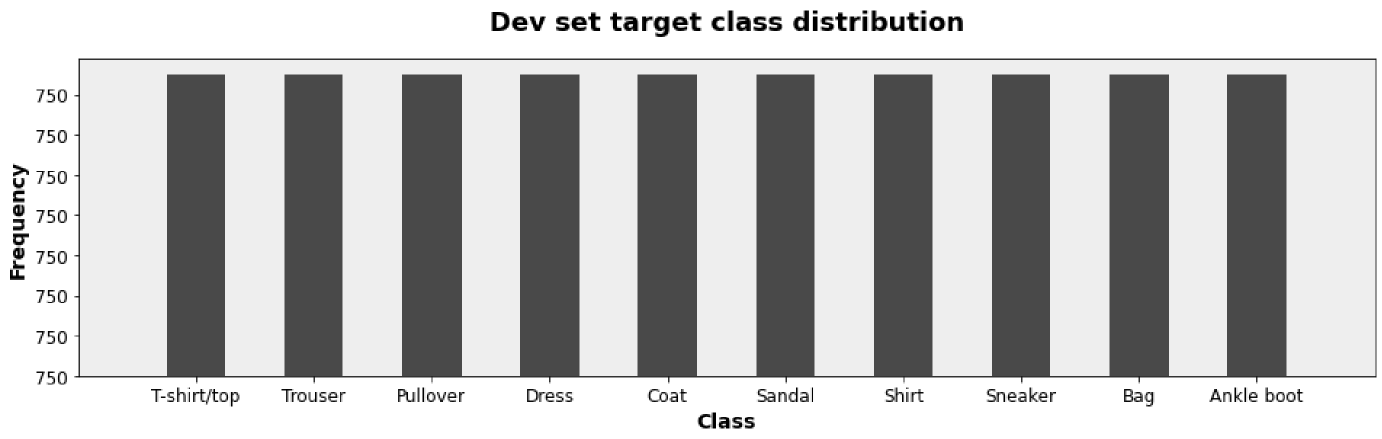
Splitting data into train / dev / test sets

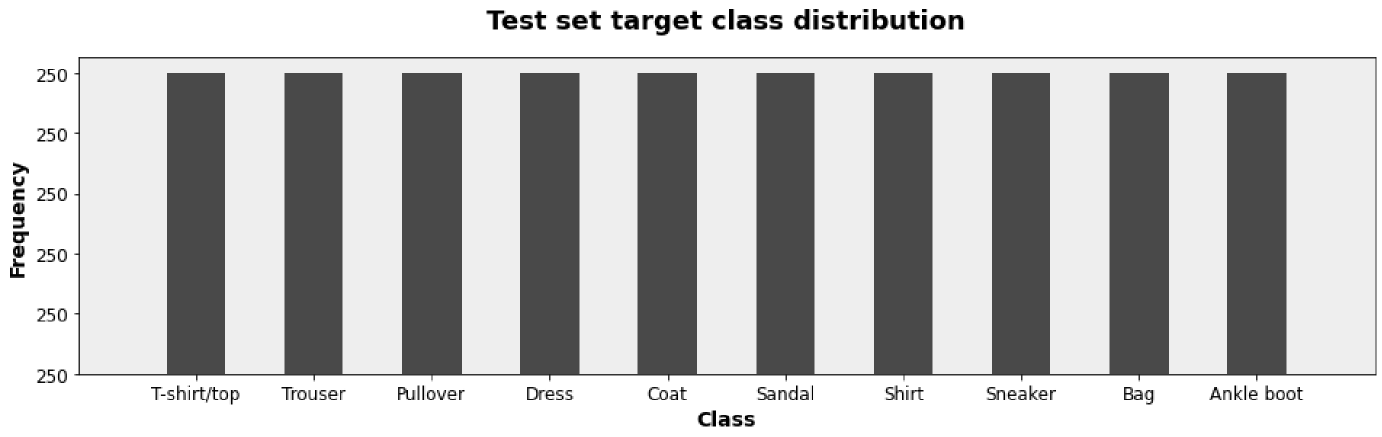
Since our dataset has already been split to train / test sets (*(X_train, y_train)*, *(X_test, y_test)* respectively), we shall proceed further by retaining a percentage (0.20) of the test data for creating a *development* set. The *development* set will be used for tuning hyperparameters of the model architectures, so as to acquire a robust model for inference.

We shall also be extra careful to retain the uniform nature of the target class distribution so as not to introduce bias. To do this, we shall split the sets in a stratified fashion.

The final splitting of our sets is as follows:

Set	Shape
X_train	(48000, 784)
y_train	(48000,)
X_dev	(12000, 784)
y_dev	(12000,)
X_test	(10000, 784)
y_test	(10000,)





General remarks / architectural choices

Model capacity

Commenting on the neural net depth, breadth and architecture (*MLP*s vs. *CNN*s), how to we know which size of network to create? A *MLP* with at least a hidden layer and a non-linearity is a universal continuous function approximation. However, creating a shallow *MLP* is trying to model the function in low vector space. Empirical observations show that deeper neural architectures work better in real problems. In practice it is often the case that 3-layer *MLP*s will outperform 2-layer *MLP*s, but going even deeper (4, 5, 6-layer) rarely helps much more.

This is in stark contrast to *CNN*s for the task of image classification, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of 10 learnable layers). One argument for this observation is that images contain hierarchical structure so several layers of processing make intuitive sense for this domain.

(source [1]).

Recall that model capacity is the ability of our model to optimaly represent the problem at hand. A lower capacity model with fail to fit its parameters during training time (i.e. the model will *underfit* the data), while an excesive capacity model will fail to generalize (i.e. it will *overfit* to the training instances).

A good illustration of the *underfitting* / *overfitting* problem is given below. Note that the key diagnostic for estimating our model's capacity to generalize is provided by the gap in the plot of *loss* / *validation loss* values rolled out over training epochs. A succint model's (i.e. a model having just-right capacity) tell-tale graph is such that both metrics diminish in accord over training iterations and their gap is reasonably wide. We deduce the evaluation of the capacity of our model by evaluating the *error* metric on both the *training* and *test* sets. The sweet spot of *underfiting* vs. *overfitting* occurs at the equilibrium of the *train error rate* vs. *test error rate*. In other words, **the optimal model capacity appears at the point at which the gap in the graph of both error metrics is minimized**.

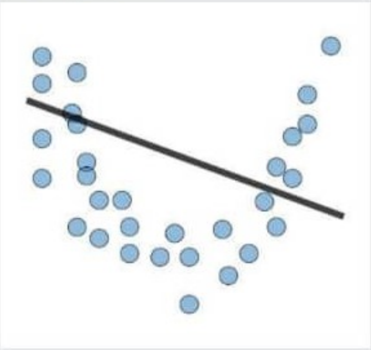
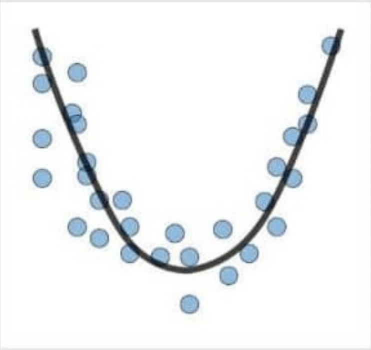
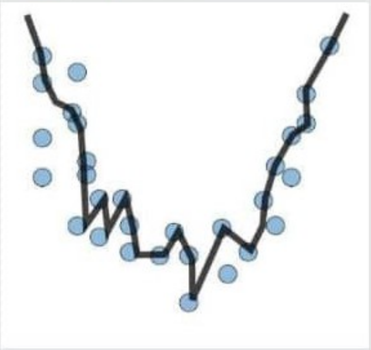
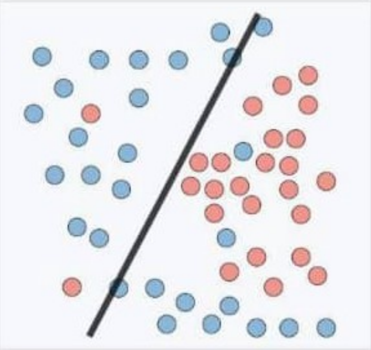
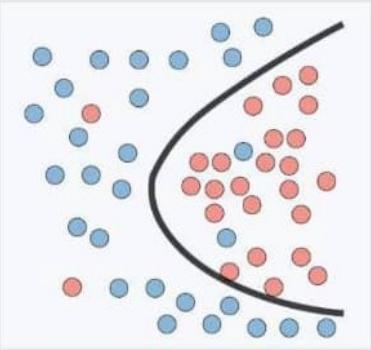
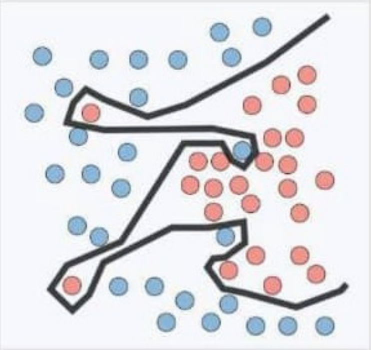
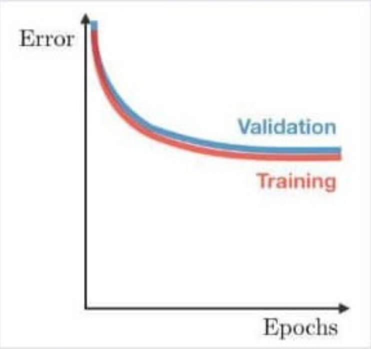
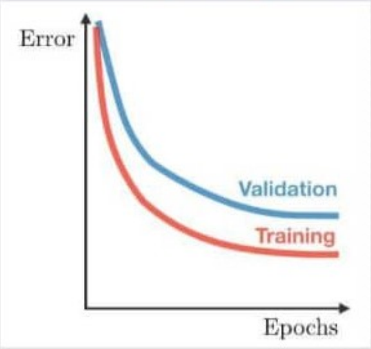
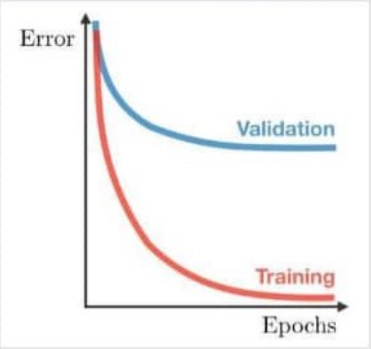
	Underfitting	Just right	Overfitting
Symptoms	• High training error • Training error close to test error • High bias	• Training error slightly lower than test error	• Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	• Complexify model • Add more features • Train longer		• Perform regularization • Get more data

Image source: [2]

Loss function

We are dealing with a multiclass classification task. For this reason, the appropriate loss function to use, regardless of network architecture (*MLP* or *CNN*), is **categorical cross-entropy** which is defined as:

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where:

$\mathbf{w}$: the model parameters

y_i: the **true** class label

$\hat{y}_i$: the **predicted** class label

N: the number of classes

Specifically, in TensorFlow terms, we shall use the built-in **sparse categorical cross-entropy** function, which provides users with the means of measuring the the loss without having to convert target variables to their one-hot representation.

Weights initialization

Choosing a weights initialization method in deep learning is critical because there is a risk that training progress will be slowed to the point at which the network fails to learn the parameters. This, in conjunction with the choice of a non-linear activation function (e.g. a *sigmoid*), happens because weights can be "pushed" to the activation function's extreme regions (e.g. at the far left or right of a *sigmoid* function), having the derivative approach zero, thereby leading to back-propagated drivatives also close to zero, thus impeding learning.

Let's illustrate the problem of *vanishing* and *exploding* gradients with an example. Please refer to the acompanying notebook "*homework-001-vanishing-gradients.ipynb*" and code for implementation.

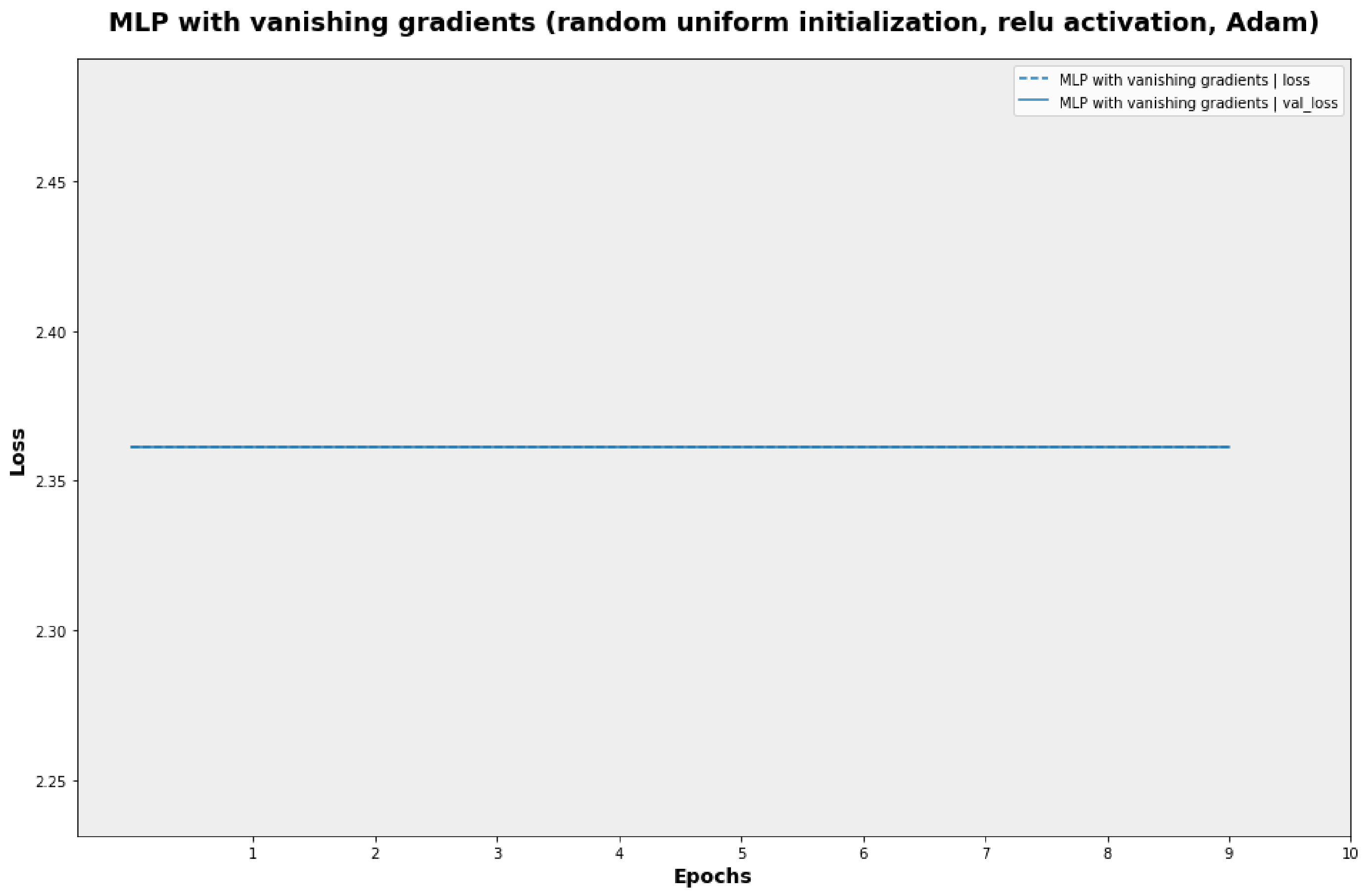
We initially showcase the problem by simulating the initialization of weights from a random uniform distribution. In the sequel, we perform 100 iterations of matrix multiplication to simulate the backward pass in a neural network. The results we get are as follows:

- *Vanishing* gradients
  - Parameters: matrices initialised in the range $[0, 0.1]$
  - Result: weights $\mathrm{mean} = (0.0,), \mathrm{std} = 0.0$
- *Exploding* gradients
  - Parameters: matrices initialised in the range $[0, 10]$
  - Result: weights $\mathrm{mean} = (inf,), \mathrm{std} = nan$

In the sequel, we selectively showcase the problem of *vanishing* gradients with training a deep-and-thin model with the following parameters:

```
Model: "fashion-mnist-mlp-vanishing-gradients1591214468"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 784)]             0
_________________________________________________________________
hidden_layer_1 (Dense)       (None, 8)                 6280
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 8)                 72
_________________________________________________________________
hidden_layer_3 (Dense)       (None, 8)                 72
_________________________________________________________________
hidden_layer_4 (Dense)       (None, 8)                 72
_________________________________________________________________
hidden_layer_5 (Dense)       (None, 8)                 72
_________________________________________________________________
hidden_layer_6 (Dense)       (None, 8)                 72
_________________________________________________________________
output_layer (Dense)         (None, 10)                90
=================================================================
Total params: 6,730
Trainable params: 6,730
Non-trainable params: 0
```

The training graph clearly showcases the problem which is evident by observing that the *loss* of the network does not diminish during training.



Current empirical practice in *CNN*s suggests that *He* initialization $w = \mathrm{np.random.randn(n)} \times \mathrm{sqrt(2.0\ /\ n)}$ is the best candidate, as discussed in [3]. In their 2015 paper, they demonstrated that deep networks would converge much earlier if the weights initialization followed their strategy.

Non-linearities

The choice of activation function is a deciding factor in neural net performance. It has been demonstrated that using the *sigmoid* as the activation function can quickly lead to a vanishing gradients problem, since

Current practice in *CNN*s suggests that *ReLU* units should be used as non-linearities, accompanied by *He* initialization $w = \mathrm{np.random.randn(n)} \times \mathrm{sqrt(2.0\ /\ n)}$, as discussed in [3]

Output layer activation function

Since our task is a *multi-class classification problem*, the activation function of the output layer is a *softmax*.

Batch size

Empirical observation leads to the conclusion that *batch_size* affects overall training performance. A very high *batch_size* will affect model accuracy, while a very small batch_size will have a negative impact on convergence. We shall experiment with a base batch size of 32 and in the sequel we shall tune the batch size during hyperparameter optimization.

Regularization

The predominant method of regularization in neural net architectures is the introduction of *dropout* layers in between the hidden layers of the architecture. *Dropout* layers enforce random non-selection of weights, up to a configurable percentage of all weights it the current layer (i.e. a hyperparameter), thereby making sure that the model does not learn some parameters at random, effectively avoiding overfitting.

Optimization algorithms

We shall experiments with *Stochastic Gradient Decent (SGD)* and *Adam*. For *SGD*, we shall also use momentum and configure it such that it is of type *Nesterov*.

Hyperparameter search / optimization

In contrast to classic machine learning models, one major challenge that one faces while working with deep learning is that there are lot of parameters to hyper tune. Hence it becomes important to appropriately select correct parameters so as to avoid overfiting and underfitting.

To tackle the hyperparameter space search problem, we shall use a Python package named *Talos* (https://github.com/autonomio/talos (https://github.com/autonomio/talos)). The package provides all the mechanics for hyperparameter tuning. Hyperparameters are provided in the form of dictionary key-value pairs.

Experiments

MLPs

Comparing shallow, medium and deep MLPs

So, which is the right architecture for the task at hand? We have established that the *ReLU* / *He* pair should be employed, however what about the network depth and breadth?

Next up, we create 3 MLPs with the following architectures / characteristics to try to answer such questions:

- A small MLP (3 hidden layer, 64 units)
- A medium MLP (5 hidden layer, 64 units)
- A large MLP (7 hidden layer, 64 units)

with the following architectures:

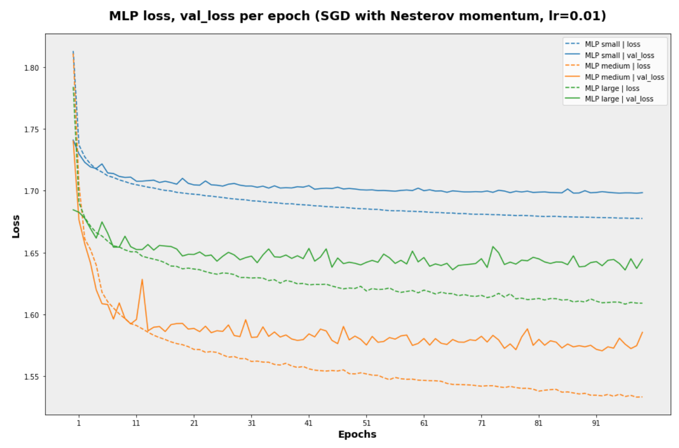
```
Model: "fashion-mnist-mlp-small-he-relu-1591214680"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 784)]             0
_________________________________________________________________
hidden_layer_1 (Dense)       (None, 128)               100480
_________________________________________________________________
output_layer (Dense)         (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

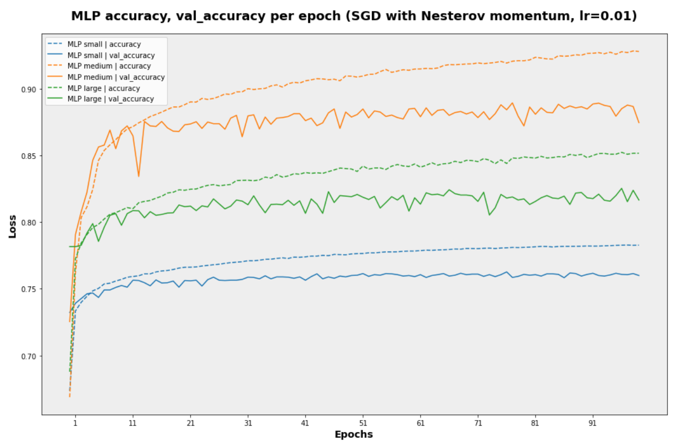
```
Model: "fashion-mnist-mlp-medium-he-relu-1591214681"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 784)]             0
_________________________________________________________________
hidden_layer_1 (Dense)       (None, 128)               100480
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 128)               16512
_________________________________________________________________
output_layer (Dense)         (None, 10)                1290
=================================================================
Total params: 118,282
Trainable params: 118,282
```

```
Non-trainable params: 0
_________________________________________________________________


Model: "fashion-mnist-mlp-large-he-relu-1591214682"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_layer (InputLayer)     [(None, 784)]             0         
_________________________________________________________________
hidden_layer_1 (Dense)       (None, 128)               100480    
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 128)               16512     
_________________________________________________________________
hidden_layer_3 (Dense)       (None, 128)               16512     
_________________________________________________________________
output_layer (Dense)         (None, 10)                1290      
=================================================================
Total params: 134,794
Trainable params: 134,794
Non-trainable params: 0
_________________________________________________________________
```

The results obtained after training for 100 epochs, using *SGD* as our optimizer with parameters $\mathrm{lr} = 0.01$ and *Nesterov momentum*, are as follows:

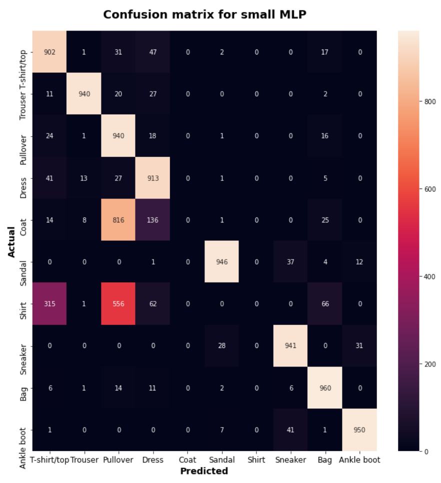


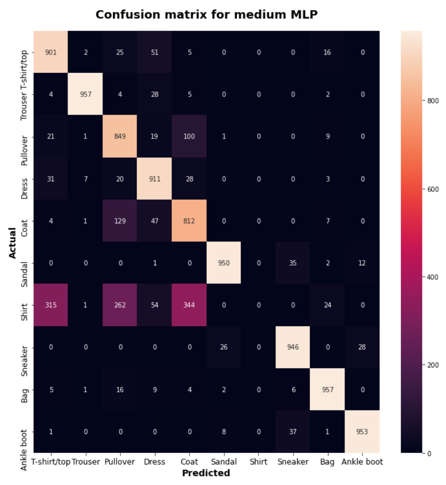


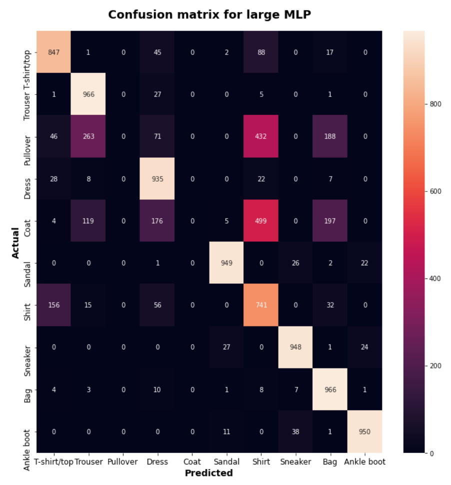
Evaluation on the test set produces the following results:

Architecture	Test loss	Test accuracy
mlp_small	1.715313	0.7492
mlp_medium	1.638555	**0.8236**
mlp_large	1.728768	0.7302

The clear winner is model **mlp_medium** which obtains the lowest *loss* / *val_loss*, highest *accuracy* as well as exhibits good properties in the trend of corresponding training curves. Visual inspection of the confusion matrix of the classifier indicates that the classifier does a decent job at getting true examples right (the diagonal of the matrix), however still misclassifies quite a few examples.





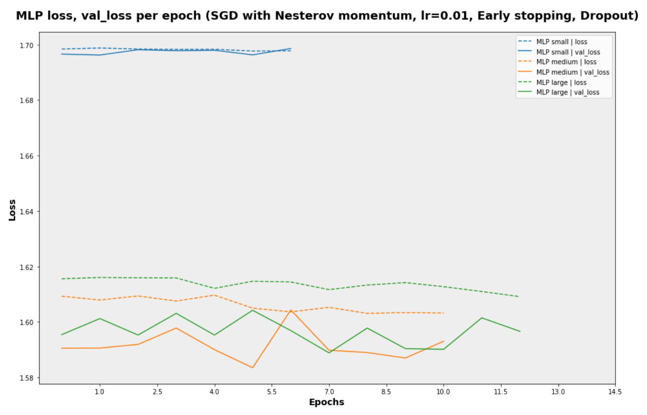


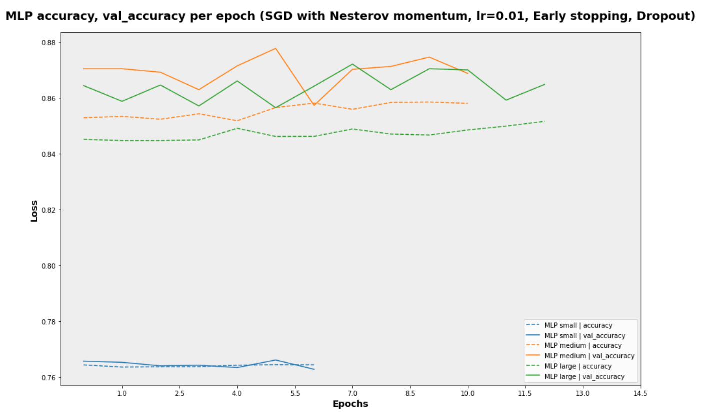
Manually improving on the baseline architecture

Our next iteration involves further measures to avoid *overfitting*, namely *regularization* and *early stopping*.

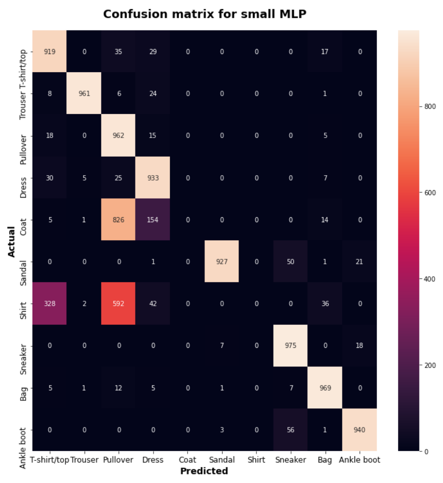
Regularization is achieved by introducing *dropout* layers in the MLP architecture. *Early stopping* monitors *loss*, *val_loss* for identyfying a point in time (epoch) in which *val_loss* hasn't decreased w.r.t. previous epochs.

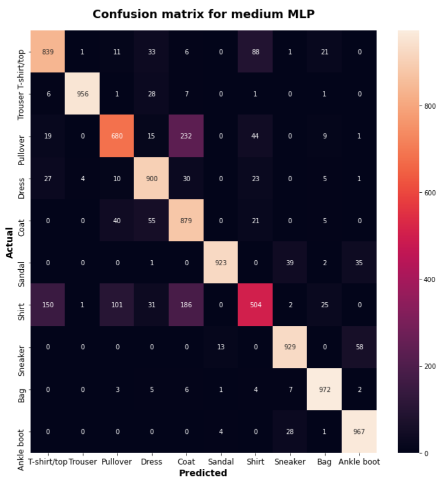
The results obtained are as follows:

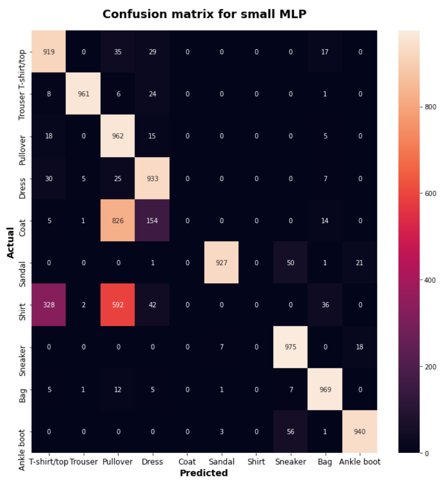




Architecture	Test loss	Test accuracy
mlp_small	1.703015	0.7586
mlp_medium	1.605595	**0.8549**
mlp_large	1.614829	0.8457







We can tell that the all models accuracy on the test set has improved significantly. The architecture previously selected, **mlp_medium**, remains the architecture of choice for the next step which involves hyper-parameter tuning.

Hyperparameter tuning of the baseline architecture

To keep the problem tractable, we explore 20% (64 iterations) of the following hyperparameter search space:

- Batch size(s): 16, 32 or 64
- Learning rate(s): 0.01 or 0.001
- Dropout rate(s): 0.1, 0.2 or 0.3
- First layer number of units: 64, 128 or 256
- Second layer nuber of units": 64, 128 or 256
- Kernel initializer method(s): *He uniform*
- Activation function(s): *ReLU*
- Optimizer(s): *SGD* with *Nesterov momentum* (0.9) or *Adam*
- Epochs: 100 (with early stopping)

We obtain the best model, found at iteration 50, with the following characteristics:

loss	first_layer_units	epochs	activation	dropout
50	1.929874	256	100	<function relu at 0x00000183E52969D8>

val_loss	optimizer	end	acc	duration
0.1	1.940557	<tensorflow.python.keras.optimizer_v2.gradient...	06/05/20-114906	0.887812

second_layer_units	val_acc	round_epochs	start	kernel_initializer	batch_size
137.369504	128	0.876250	56	06/05/20-114649	<tensorflow.python.ops.init_ops_v2.VarianceSca...

which yields a validation accuracy score of 0.869 and test set accuracy at 0.86.

CNNs

Convolutional neural networks are a flavor of neural nets tailored for the task of image of performing deep learning on input data that are, usually, images. This assumption allows us to encode certain properties into the architecture.

Having exhaustively gone through the process of training different *MLP*s, we shall proceed with the creation of 3 baseline *CNN*s with the following architectures / characteristics:

- A small *CNN* (1 convolutional layer, 32 filters)
- A medium *CNN* (2 convolutional layers, 32 / 64 filters)
- A large *CNN* (3 convolutional layer, 32 / 64 / 128 filters)

with the following architectures:

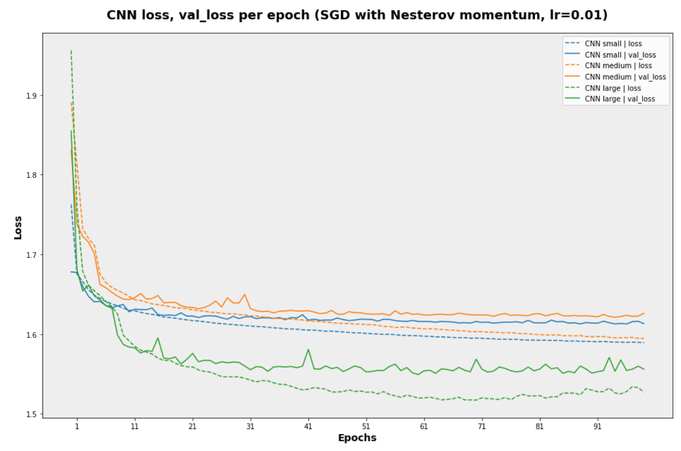
```
Model: "fashion-mnist-cnn-small-relu-1591677112"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 28, 28, 1)]       0
_________________________________________________________________
conv_2d_layer_1 (Conv2D)     (None, 28, 28, 32)        320
_________________________________________________________________
max_pool_2d_layer_1 (MaxPool (None, 14, 14, 32)        0
_________________________________________________________________
flatten_layer (Flatten)      (None, 6272)              0
_________________________________________________________________
output_layer (Dense)         (None, 10)                62730
=================================================================
Total params: 63,050
Trainable params: 63,050
Non-trainable params: 0
_________________________________________________________________
```

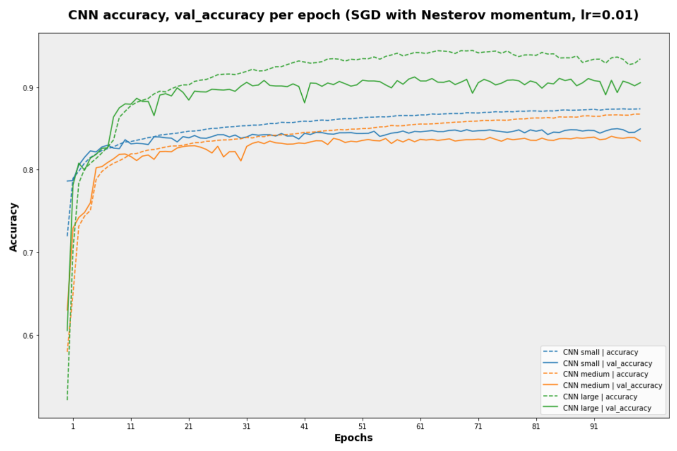
```
Model: "fashion-mnist-cnn-medium-relu-1591677114"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 28, 28, 1)]       0
_________________________________________________________________
conv_2d_layer_1 (Conv2D)     (None, 28, 28, 32)        320
_________________________________________________________________
max_pool_2d_layer_1 (MaxPool (None, 14, 14, 32)        0
_________________________________________________________________
conv_2d_layer_2 (Conv2D)     (None, 14, 14, 64)        18496
_________________________________________________________________
max_pool_2d_layer_2 (MaxPool (None, 7, 7, 64)          0
_________________________________________________________________
flatten_layer (Flatten)      (None, 3136)              0
_________________________________________________________________
```

```
output_layer (Dense)         (None, 10)                31370
=================================================================
Total params: 50,186
Trainable params: 50,186
Non-trainable params: 0
_________________________________________________________________
```

```
Model: "fashion-mnist-cnn-large-relu-1591677114"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 28, 28, 1)]       0
_________________________________________________________________
conv_2d_layer_1 (Conv2D)     (None, 28, 28, 32)        320
_________________________________________________________________
max_pool_2d_layer_1 (MaxPool (None, 14, 14, 32)        0
_________________________________________________________________
conv_2d_layer_2 (Conv2D)     (None, 14, 14, 64)        18496
_________________________________________________________________
max_pool_2d_layer_2 (MaxPool (None, 7, 7, 64)          0
_________________________________________________________________
conv_2d_layer_3 (Conv2D)     (None, 7, 7, 128)         73856
_________________________________________________________________
max_pool_2d_layer_3 (MaxPool (None, 4, 4, 128)         0
_________________________________________________________________
conv_2d_layer_4 (Conv2D)     (None, 4, 4, 256)         295168
_________________________________________________________________
max_pool_2d_layer_4 (MaxPool (None, 2, 2, 256)         0
_________________________________________________________________
flatten_layer (Flatten)      (None, 1024)              0
_________________________________________________________________
output_layer (Dense)         (None, 10)                10250
=================================================================
Total params: 398,090
Trainable params: 398,090
Non-trainable params: 0
_________________________________________________________________
```

The results obtained after training for 100 epochs, using *SGD* as our optimizer with parameters $\mathrm{lr} = 0.01$ and *Nesterov momentum*, are as follows:



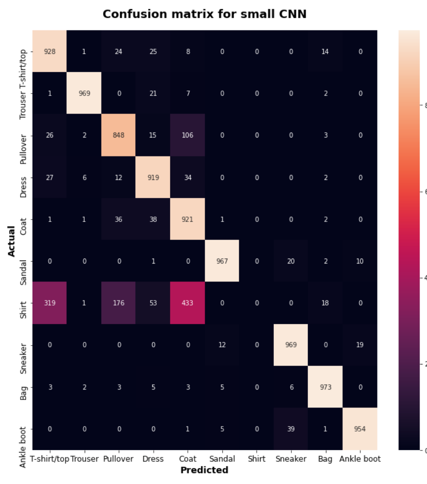


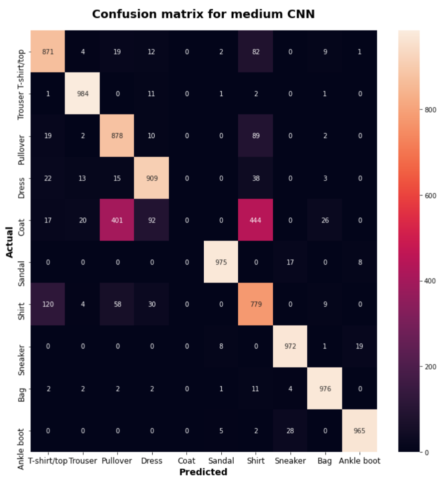
Evaluation on the test set produces the following results:

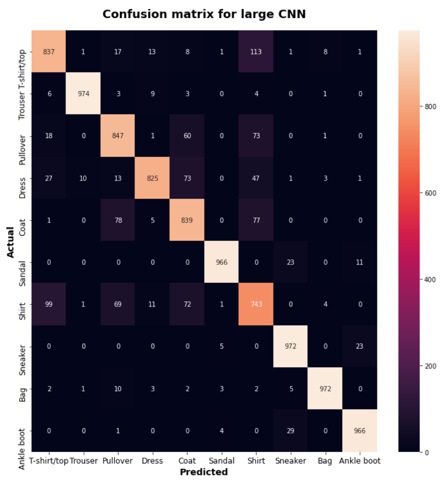
Architecture	Test loss	Test accuracy
cnn_small	1.615897	0.8448
cnn_medium	1.629447	0.8309
cnn_large	1.566829	0.8941

This time around, it is evident that the larger capacity of a *CNN* is beneficial, as the larger architecture produces the best results on the test set and achieves 0.894 accuracy.

Let's also take a look at the corresponding confusion matrices:



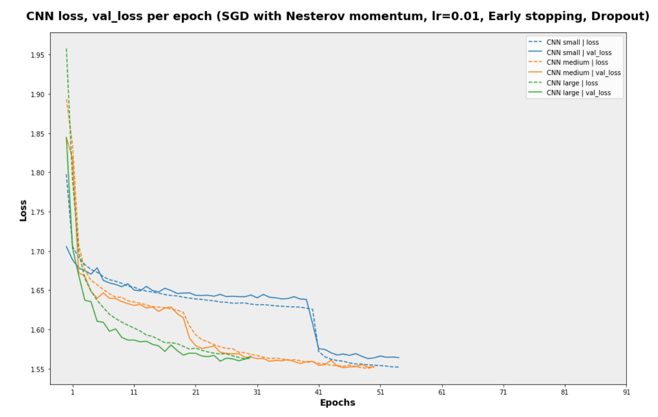


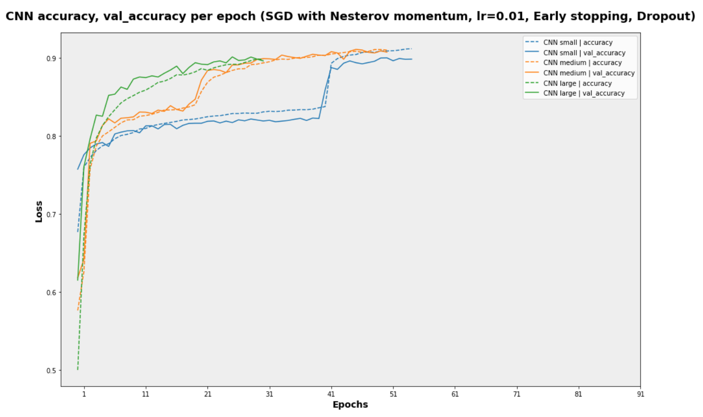


Manually improving on the baseline architecture

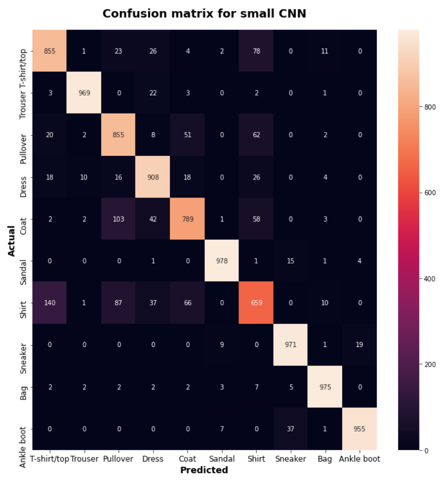
Same as with the *MLP*s our next iteration involves further measures to avoid *overfitting*, again by introducing *dropout* layers in the architecture.

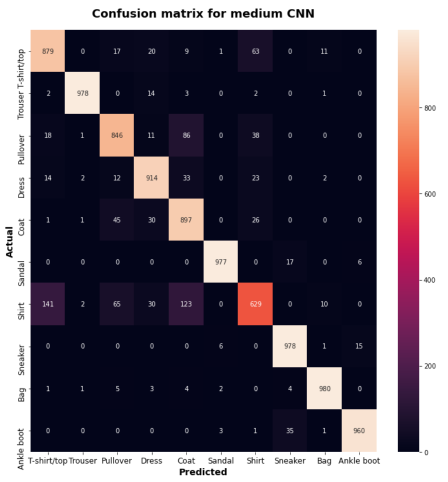
The results obtained are as follows:

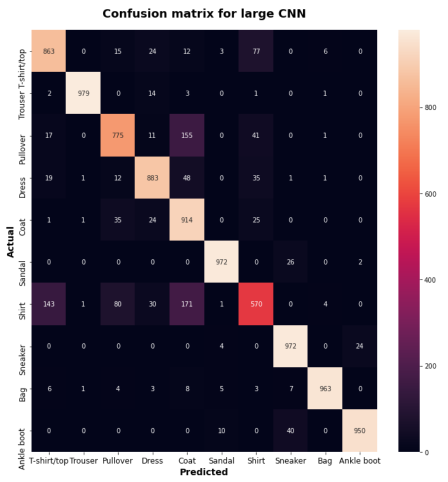




Architecture	Test loss	Test accuracy
cnn_small	1.571237	0.8914
cnn_medium	1.558384	0.9038
cnn_large	1.576308	0.8841







What is interesting is that early stopping kicked-in at later epochs, when compared to the *MLP*s. Also, the medium-sized architecture performs better this time, obtaining ac accuracy score of 0.90. We shall therefore continue on improving this architecture by performing hyperparameter tuning in the sequel.

Hyperparameter tuning of the baseline architecture

Keeping hyperparameter space in check for outr current scope is a balancing exercise. It is easy for the search space to explode in the order of hundreds of thousands of iterations. Therefore, we shall limit the parameters for which to optimize in the following set:

- Batch size: we shall keep a fixed 32 batch size parameter, as obtained from the *MLP* case.
- Learning rate(s): 0.01 or 0.001
- Dropout rate(s): 0.1, 0.2 or 0.3
- Convolutional filters: 32, 64 or 128
- Convolutional kernel sizes: 3×3 or 5×5
- Striding: 1 or 2 spaced
- Padding: fixed to *'same'*

- Dilation rate: fixed to 1
- Max-pool size: fixed to $(2, 2)$
- Activation function(s): fixed to *ReLU*
- Optimizer(s): *SGD* with *Nesterov momentum* (0.9) or *Adam*
- Epochs: 100 (with early stopping)

Furthermore, to keep the problem tractable, we explore only 10% $(172$ iterations) of the previously described space. Given more time / computig resources we could, certainly, explore more possibilities.

We obtain the following best parameters:

learning_rate	batch_size	optimizer	end	output_activation
0.010	32	<tensorflow.python.keras.optimizer_v2.gradient...	06/09/20-144814	<function relu at 0x00000286C1FC79D8>

duration	loss	round_epochs	val_acc	first_layer_conv_dilation_rate
125.127403	0.142602	11	0.922083	(1, 1)

second_layer_activation	second_layer_conv_filters	start	dropout	second_layer_maxpool_strides
relu	64	06/09/20-144609	0.1	(1, 1)

first_layer_maxpool_pool_size	first_layer_conv_filters	epochs	second_layer_maxpool_padding	val_loss
(2, 2)	64	100	same	0.243917

first_layer_conv_kernel_size	first_layer_maxpool_padding	second_layer_conv_strides	second_layer_conv_kernel_si
(3, 3)	same	(1, 1)	(3,

second_layer_conv_dilation_rate	second_layer_maxpool_pool_size	acc	first_layer_conv_padding
(1, 1)	(2, 2)	0.947771	same

first_layer_maxpool_strides	second_layer_conv_padding	first_layer_conv_strides	first_layer_activation
(1, 1)	same	(1, 1)	relu

which yields a validation accuracy score of 0.922 and a test accuracy score of 0.916.

References

1. [https://cs231n.github.io/neural-networks-1/] (https://cs231n.github.io/neural-networks-1/%5D) CS231n Convolutional Neural Networks for Visual Recognition

2. [https://medium.com/@sayedathar11/multi-layered-perceptron-models-on-mnist-dataset-say-no-to-overfitting-3efa128a019c] (https://medium.com/@sayedathar11/multi-layered-perceptron-models-on-mnist-dataset-say-no-to-overfitting-3efa128a019c%5D) Multi-Layered-Perceptron Models on MNIST Dataset : Say No to Overfitting!

3. [https://arxiv.org/abs/1502.01852] (https://arxiv.org/abs/1502.01852%5D) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification